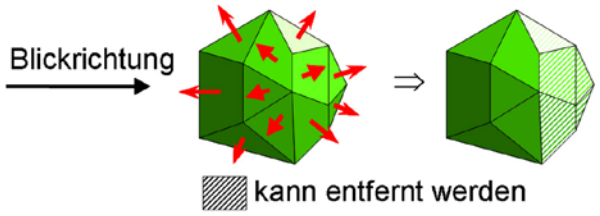
9. Sichtbarkeitsverfahren

Um Szenen glaubwürdig und korrekt darzustellen, müssen alle Teile, die aus der Blickrichtung nicht sichtbar sind, weggelassen werden. Das sind insbesondere die Rückseiten von Objekten und Objektteile, die von anderen Objekten verdeckt werden. Man spricht von Hidden-Line- oder Hidden-Surface Eliminierung.

Abhängig von der Szenenkomplexität, den Objekttypen und -datenstrukturen, der verfügbaren Hardware und Anwendungsanforderungen verwendet man dazu verschiedene *Sichtbarkeitsverfahren*. Bei den Objektraum-Methoden wird die Lage der Objekte miteinander verglichen und nur vordere (sichtbare) Teile gezeichnet, bei den Bildraum-Methoden wird für jeden Bildteil getrennt berechnet, was dort sichtbar ist. Die folgenden Erklärungen erfolgen ohne Berücksichtigung von transparenten Objekten.

Backface Detection (Backface Culling)

Backface Culling ist ***kein*** vollständiges Sichtbarkeits-verfahren. Es werden lediglich alle Polygone, deren Oberflächennormale vom Betrachter weg zeigt und die daher ganz sicher nicht sichtbar sein können, eliminiert, um den Aufwand nachfolgender Arbeitsschritte zu reduzieren. Dadurch werden im Durchschnitt 50% der Polygone entfernt. Dies berechnet man bei orthographischer Projektion mit dem Skalarprodukt des Blickrichtungsvektors mit der Oberflächennormale ($V_{view} \cdot N > 0 \longrightarrow$ unsichtbar) oder bei perspektivischer Projektion durch Einsetzen des Blickpunktes (x,y,z) in die Ebenengleichung ($Ax + By + Cz + D < 0 \longrightarrow$ unsichtbar). [Annahmen wie bei „Polygonlisten“].



Z-Puffer-Verfahren (Depth Buffer)

Der z-Puffer-Algorithmus löst das Sichtbarkeitsproblem für eine bestimmte Bildauflösung folgendermaßen: Für jeden Bildpunkt merkt man sich zusätzlich zur Farbinformation in einem eigenen Speicher die Position des dargestellten Objektes. Da als Blickrichtung normalerweise die z-Richtung verwendet wird, entsprechen x- und y-Werte dieser Position denen der Abbildungsebene und es braucht nur der z-Wert gespeichert zu werden. Man braucht also zusätzlich zum Bildpuffer (frame buffer) einen weiteren Speicherbereich, der für jedes Pixel einen Koordinatenwert (z-Wert) aufnehmen kann, diesen Speicher nennt man z-Puffer oder Tiefenpuffer (z-buffer, depth buffer). Nun kann man alle Objekte in beliebiger Reihenfolge zeichnen. Die z-Werte des nächsten zu zeichnenden Objektes (meist ein Polygon) werden berechnet und mit den z-Werten der Pixel verglichen, in die das Objekt gezeichnet werden soll. Ist der neue z-Wert näher zum Betrachter (also normalerweise größer), dann wird das Objekt an dieser Stelle über den alten Bildwert darüber gezeichnet und der z-Wert im z-Puffer ebenfalls ersetzt. Andernfalls ist das neue Objekt verdeckt und wird an dieser Stelle nicht gezeichnet:

```
for all (x,y)  (* Initialisierung des Hintergrundes *)
  depthBuff(x,y) = -1  (* größtmögliche Entfernung *)
  frameBuff(x,y) = backgroundColor
for each polygon P   (* Schleife über alle Polygone *)
  for each position (x,y) on polygon P
    calculate depth z
    if z > depthBuff(x,y) then
      depthBuff(x,y) = z
      frameBuff(x,y) = surfColor(x,y)    (* else nichts ! *)
```

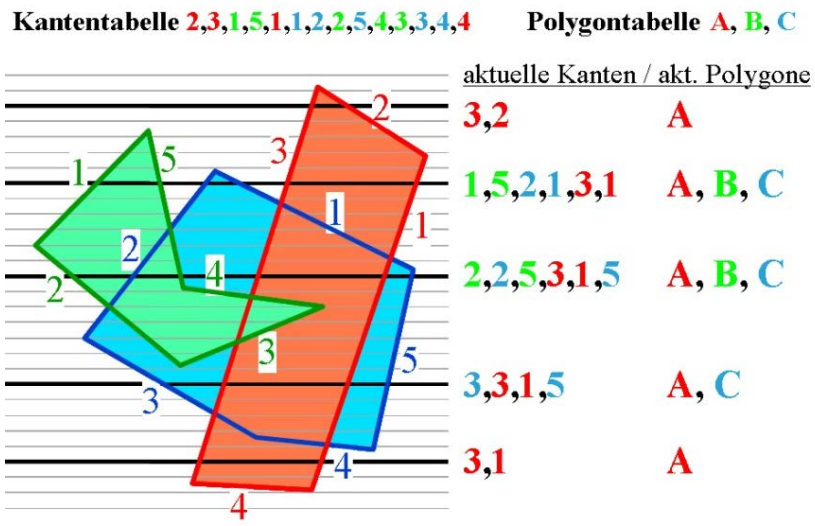
Für ebene Polygone lassen sich die z-Werte natürlich wieder inkrementell effizienter berechnen. Der große Vorteil des z-Puffer-Verfahrens ist, dass die Objekte (Polygone) nicht sortiert werden brauchen.

(Anmerkung: Bei der Viewport-Transformation wird oft z mit –1 multipliziert bevor das z-Puffer-Verfahren angewendet wird.)

Scanline-Methode

Die korrekte Sichtbarkeit wird beim Scanline-Verfahren zeilenweise berechnet (im Beispiel von oben nach unten, also y fallend). Dabei nutzt man aus, dass sich zwei übereinander liegende Pixelreihen (Scanlines) in ihrem Sichtbarkeitsverhalten oft nur unwesentlich unterscheiden.

Ausgehend von einer Tabelle aller nach ihrem größten y-Wert sortierten Polygonkanten und einer zugehörigen Polygonliste wird für jede Scanline jeweils eine Liste der aktuellen Kanten erstellt. Dies passiert inkrementell aus den Kanten der letzten Scanline: Kanten, die geendet haben, werden eliminiert, und die nächsten Kanten der sortierten Tabelle werden überprüft, ob sie schon angefangen haben. Nachdem solcherart alle Schnittpunkte einer Scanline mit allen Kanten errechnet sind, werden diese nach x-Wert (von links nach rechts) sortiert. Zwischen je zwei Schnittpunkten muss nun noch eruiert werden, welches Polygon dort am nächsten zum Betrachter liegt, dieses ist sichtbar und wird entlang dieser einen Scanline gezeichnet.



Depth-Sorting-Methode

Das Grundprinzip des Depth-Sorting-Verfahrens ist es, alle Polygone von hinten nach vorne zu sortieren und diese anschließend in dieser Reihenfolge zu zeichnen. Da alle verdeckten Teile weiter hinten sind als die sie verdeckenden Teile, bleibt am Ende ein Bild in korrekter Sichtbarkeit über (*painter's algorithm*). Der Hauptaufwand liegt hier beim Sortieren, das so erfolgen muss, dass kein Polygon ein anderes (partiell) verdeckt, welches in der Liste erst nachher kommt (also weiter „vorne" liegt). Dazu wird zuerst schnell eine Grobsortierung durchgeführt und anschließend überprüft, ob alles stimmt, gegebenenfalls umsortiert.

1. Grobsortierung: ordne die Polygone nach ihrem kleinsten z-Wert (größte Tiefe)
2. Vergleiche jedes Polygon S mit jedem (!) anderen Polygon S': (* Annahme S liegt hinter S' *)

(* jetzt beginnt eine Folge von Tests in aufsteigender Komplexität, bis Sortierung als korrekt erkannt ist *)

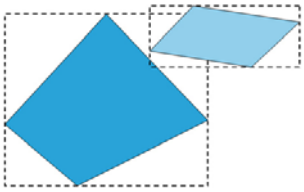
a. der größte z-Wert von S ist kleiner als der kleinste z-Wert von S' ⟶ Sortierung korrekt

b. die x- oder y-Intervalle der beiden Polygone schneiden sich nicht ⟶ Sortierung korrekt

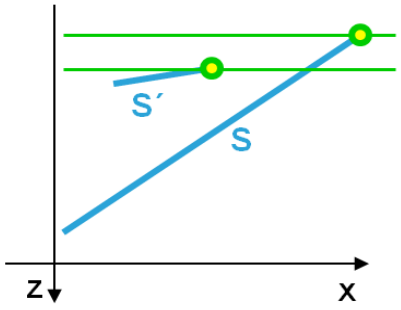
c. alle Eckpunkte von S liegen hinter Ebene von S' ⟶ Sortierung korrekt

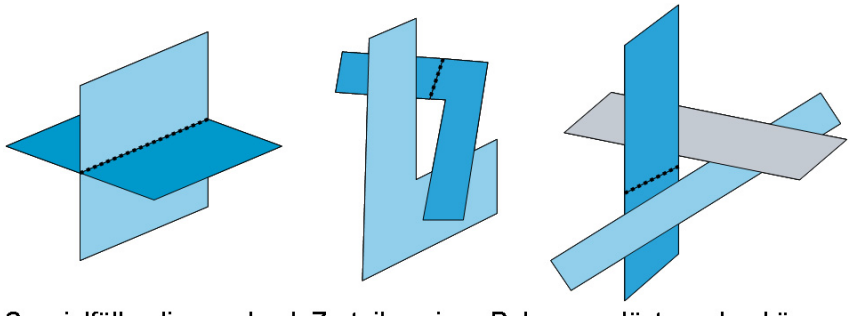
d. alle Eckpunkte von S' liegen vor der Ebene von S ⟶ Sortierung korrekt

e. die Projektionen von S und S' auf die xy-Ebene schneiden sich nicht (siehe Bild rechts) ⟶ Sortierung korrekt



f. Sortierung wahrscheinlich falsch (siehe etwa das Beispiel, wo S' von S verdeckt wird) ⟶ Vertauschen und neu kontrollieren; sollte die Sortierung wieder falsch sein, dann liegt ein Spezialfall vor (siehe Abbildung) und muss durch Zerteilen eines Polygons gelöst werden:



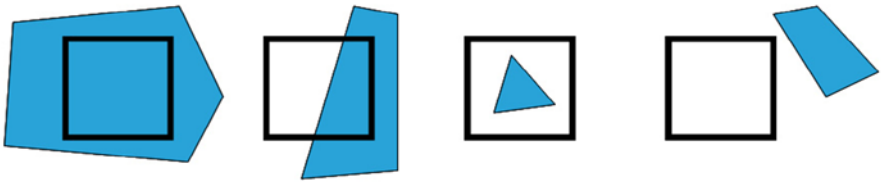


Spezialfälle, die nur durch Zerteilen eines Polgons gelöst werden können

Area-Subdivision Methode

Ähnlich wie bei der Quadtree-Repräsentation von Bildern werden auch hier einfache Fälle in grober Auflösung gelöst und kompliziertere Fälle durch Unterteilung der Fläche in vier Viertel vereinfacht. Rekursive Anwendung bis maximal zur Bildauflösung garantiert eine pixelgenaue Lösung des Sichtbarkeitsproblems.

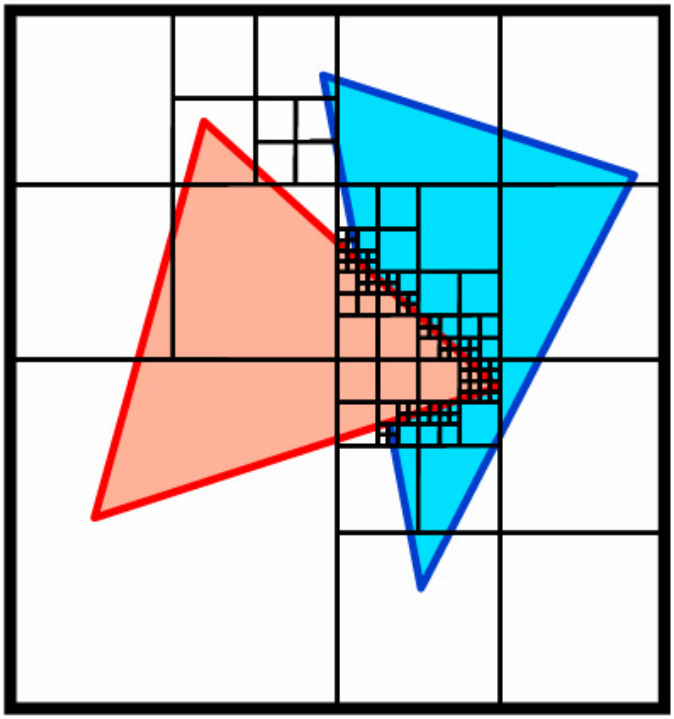
Basis des Verfahrens ist eine Funktion, die schnell feststellt, in welcher Lage sich die Projektion eines Polygons bezüglich eines (quadratischen) Bildfensters befindet. Es gibt nur 4 Möglichkeiten:



(1) das Polygon überdeckt das Fenster,
(2) das Polygon liegt teilweise innerhalb und teilweise außerhalb des Fensters,
(3) das Polygon liegt ganz im Fenster,
(4) das Polygon liegt ganz außerhalb des Fensters.

Nun gibt es drei einfache Sichtbarkeitsentscheidungen:

1. alle Polygone liegen außerhalb des Fensters → fertig
2. nur ein Polygon hat mit dem Fenster einen nichtleeren Schnitt → dieses Polygon zeichnen
3. ein Polygon überdeckt das Fenster und liegt vor allen anderen Polygonen im Fensterbereich → dieses Polygon zeichnen

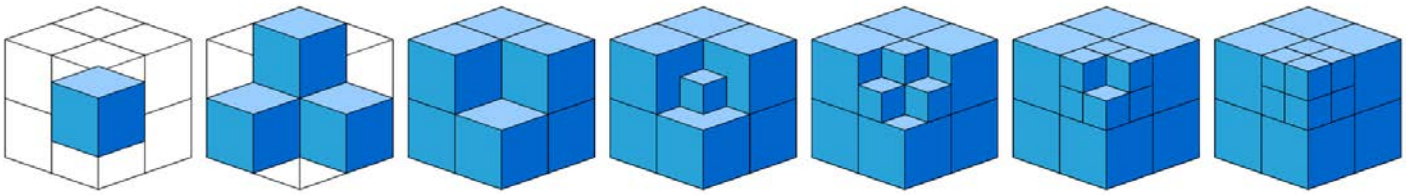


Wenn alle 3 Tests fehlschlagen, dann wird das Fenster in 4 Viertel unterteilt, und diese werden rekursiv bearbeitet. Man beachte, dass Polygone, die bereits außerhalb lagen, auch außerhalb aller Teilfenster liegen, und dass Polygone, die das Fenster überdeckt haben, auch alle Teilfenster überdecken. Wenn ein Teilfenster nur noch die Größe eines Pixels hat, so wählt man dort das Polygon, das am weitesten vorne liegt. Wie man am Beispiel sieht, passiert das genau an den Kanten, an denen die Sichtbarkeit wechselt.

Octree-Methode

Wenn die Szene statt in Polygonen als Octree repräsentiert ist, dann weiß die Datenstruktur bereits für jede Blickrichtung, was vorne und was hinten ist. Rekursiv kann man in jedem Würfel immer zuerst den entferntesten Teilwürfel rendern, dann die 3 nächstnäheren, dann die nächsten drei und schließlich den vordersten. Eine mögliche Reihenfolge für eine frontale Blickrichtung ist in dem folgenden Beispiel zu sehen:

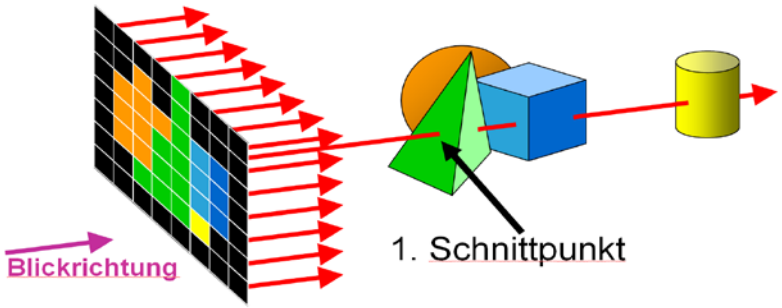
Alternativ kann man natürlich auch von vorne nach hinten zeichnen. Dann muss man sich alle Bereiche merken, in



denen bereits etwas eingetragen ist, und zeichnet nur Dinge, die auch wirklich sichtbar bleiben. Der Vorteil gegenüber anderen Datenstrukturen bleibt das implizite Wissen darüber, was weiter vorne oder weiter hinten ist.

Ray-Casting

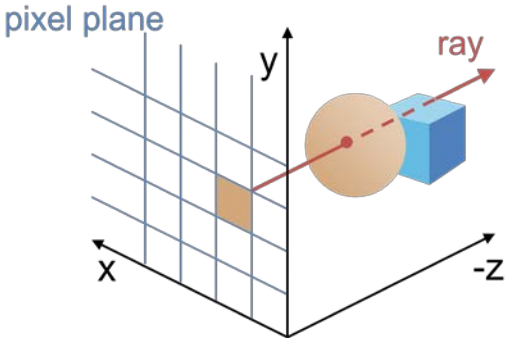
Ray-Casting ist ein Sichtbarkeitsverfahren, das für jedes Pixel getrennt berechnet, was dort sichtbar ist. Dazu wird vom Pixel aus in Blickrichtung ein *Blickstrahl* gelegt, das ist eine gerade Linie durch das Pixel in die Szene. In umgekehrter Richtung gelangt auf dieser Linie das Licht aus der Szene auf die Bildebene, also zum Betrachter. Schneidet man diesen Blickstrahl mit allen Objekten/Polygonen der Szene, so erhält man eine Menge von Schnittpunkten, aus denen man den auswählt, der zum Betrachter am nächsten liegt. Die Farbe der Oberfläche an dieser Stelle bestimmt die Farbe des Pixels, durch das man den Strahl gelegt hat. Macht man das für alle Pixel, so erhält man für jeden Punkt die Farbe des dort vordersten Objektes, also des sichtbaren Objektes.



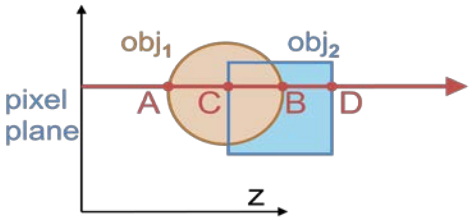
Mit Ray-Casting kann man außer Polygonen auch alle anderen Oberflächen (z.B. Freiformflächen) leicht rendern, für die der Schnitt mit einer Geraden berechenbar ist. Wie wir später sehen werden benötigt man normalerweise auch die Oberflächennormale an der Auftreffstelle, um eine brauchbare Schattierung zu erhalten. Andererseits ist das Verfahren aber sehr aufwändig, da man für jedes Pixel (das sind für einen Bildschirm einige Millionen) einen Schnitt mit jedem Objekt durchführen muss (und das können tausende bis millionen Objekte sein). Effiziente Implementierung des Schnittpunkt-Tests und weitere Optimierungen sind daher notwendig.

Ray-Casting von CSG-Objekten

Die gebräuchlichste Methode um CSG-Objekte abzubilden ist das Ray-Casting, bei dem das Bild pixelweise berechnet wird. Für jedes Pixel wird in Blickrichtung ein Strahl (Ray) gelegt („auswerfen“ = to cast) und mit allen Objekten der Szene geschnitten. Der vorderste dieser Schnittpunkte gibt an, welches Objekt in diesem Pixel zu sehen ist, und das Pixel erhält dessen Farbe. Bei einem CSG-Baum erfolgt diese Berechnung rekursiv:



- bei **Endknoten** ist die Berechnung aller Schnittpunkte einfach,
- bei **Zwischenknoten** werden die Schnittpunktlisten der beiden Nachfolger entsprechend dem Operator verknüpft:
aus den Listen (A,B) und (C,D) im Beispiel rechts entsteht
  bei Vereinigung die Liste (A,D),
  bei Durchschnitt die Liste (C,B),
  bei Differenz die Liste (A,C).
- bei der **Baumwurzel** wird der erste Punkt der verknüpften Schnittpunktliste ausgewählt.



Ray-Casting ist eine vereinfachte Version von Ray-Tracing, mit dem noch viele weitere optische Effekte simuliert werden können. Dazu kommen wir in einem späteren Kapitel.

```
Ray-Casting = für jedes Pixel der Darstellungsfläche:
  - erzeuge eine Gerade durch das Pixel in Blickrichtung („Blickstrahl“)
  - schneide den Blickstrahl mit allen Objekten
  - wähle aus der Schnittpunktliste den zum Betrachter nächsten Punkt
  - färbe das Pixel mit der Farbe der Oberfläche dieses Punktes
```

Klassifizierung der Verfahren

Wir wollen nun noch überlegen, welche Verfahren im Objektraum arbeiten und welche im Bildraum. Dies ist nicht immer ganz eindeutig klassifizierbar, aber im Großen und Ganzen gilt:

Objektraum-Verfahren: Backface Detection, Depth Sorting, Octree-Methode
Bildraum-Verfahren: Z-Puffer, Scanline-Methode, Area Subdivision, Ray Casting